Building Excel Real-Time Data Components in Visual Basic .NET

Click here to download sample - odc_xlrtdvbnet.exe. (5.29 MB)
odc_xlrtdvbnet
MSDNSamples\Excel\RTD Components

J. Sawyer
Microsoft Corporation

October 2001

Applies to:
Microsoft® Excel 2002
Microsoft® Visual Basic® .NET Beta 2

Download Odc_xlrtvbnet.exe.

Summary: This article walks you through building a real-time data server for Microsoft Excel 2002 using Visual Basic .NET Beta 2. (14 printed pages)

Contents

Introduction

The release of Microsoft® Office XP opened up a new world for developers. Smart tags have gotten all the press but they aren't the only cool new developer feature in Office. Microsoft Excel 2002, a member of the Office XP suite, introduces a way to get real-time data into your spreadsheets in an efficient, reliable, and robust manner—Excel real-time data (RTD) components. This real-time data can include cell references and can also be used in formulas, calculations, and charts.

But Office XP isn't the only thing new for developers coming from Microsoft this year. Also in the news is Microsoft® Visual Studio® .NET, a new development environment built on the Microsoft .NET initiative. Part of Visual Studio .NET is an awesome list of feature updates to the Visual Basic language, called Microsoft Visual Basic .NET.

In this article, we will walk through the process of building an RTD component in Visual Basic .NET Beta 2. Along the way, we'll look at how COM interoperability works in Visual Basic .NET and point out some of the differences between Visual Basic .NET and Visual Basic 6.0, as well as how some of the data types interact between COM and the .NET Framework class libraries. This article is intended for Visual Basic developers who are somewhat familiar with Excel 2002 RTD but are new to Visual Basic .NET. For common questions and answers regarding real-time data components, see Real-Time Data: Frequently Asked Questions. If you want to get a detailed overview of building real-time data components with Visual Basic 6.0, see Building a Real-Time Data Server in Excel 2002.

What is Excel Real-Time Data?

Excel real-time data (RTD) components are new to Excel 2002. RTD components and their accompanying infrastructure are designed to allow real-time data to be populated in an Excel spreadsheet efficiently, robustly, and without dropping an update. While there were ways to get real-time data into Excel before RTD, ranging from

Dynamic Data Exchange (DDE) to custom macros and add-ins that ran continuously in a background loop, these methods were, at best, inelegant and, at worst, unreliable and prone to stop responding. It was also difficult to drive recalculations and charts from these methods, as well as impossible to use cell references. RTD in Excel 2002 resolves these issues.

RTD servers are Component Object Model (COM) Dynamic Link Libraries (DLLs) that implement a specific interface provided by Excel 2002, the **IRTDServer** interface. The RTD server is instantiated by Excel when a user enters a new function (**RTD**) into Excel and specifies the RTD server's programmatic ID (ProgID). Once the server is instantiated, Excel gives it a reference to a callback object and communication occurs via a push-pull mechanism; that is, the RTD server notifies Excel that data has changed (push) and, when appropriate, Excel requests the changed data from the RTD server (pull). This push-pull architecture allows for RTD to function even if there is a modal dialog box displaying or if there is something else happening that would normally prevent an update.

The **IRTDServer** interface is simple and clean; the tough part can be in the implementation. However, once you've gotten the hang of it, a (very) simple RTD server can be written quickly and with minimal trouble. The challenge is then in implementing the data feed into the Excel RTD component!

Why Use .NET for Excel RTD?

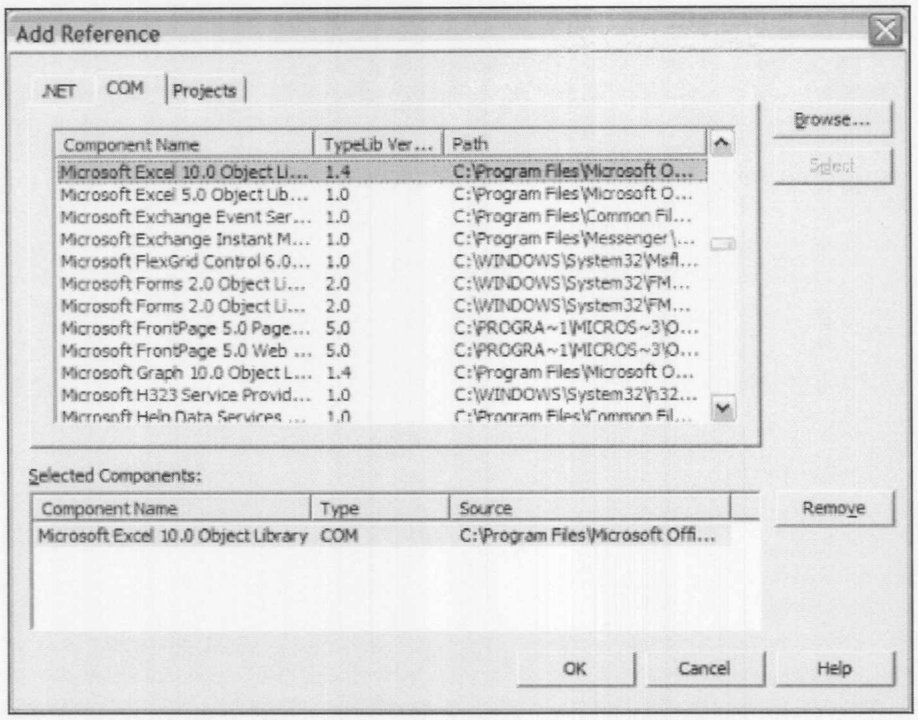
There are some good, compelling reasons for using .NET to create Excel real-time data components. First, the tight integration with .NET XML Web services gives us new avenues to explore for our real-time data feeds—imagine a stock ticker RTD component that gets stock quotes from an XML Web service or a weather RTD component that gets the latest weather from yet another XML Web service. Also, the .NET Framework class libraries expose enhanced network functionality, such as multicasting, that previously required low-level socket coding. Last, but certainly not least, you can use a higher level network API such as Microsoft DirectPlay®, which, when integrated with the .NET Framework's integrated and easy-to-use multithreading, allows you to quickly and easily design worker threads to process the incoming data while processing Excel requests for updates on another thread. In this case, it is also easy to write a true multi-threaded service in the .NET Framework for your DirectPlay server. These advanced network protocols allow you to efficiently distribute real-time data across an enterprise environment without overloading the network infrastructure.

The Visual Basic .NET RTD Server

The RTD server that we will be building will be relatively simple. It will be a server that takes stock symbols and randomly updates the stock price. It will also take an argument to return the opening price for the stock (which doesn't get updated) or the last price for the stock (which gets updated randomly). This will allow us to worry less about the code that is doing the updates and concentrate more on the implementation of the components in Visual Basic .NET. In doing this, we will look at COM interoperability in Visual Basic .NET and go over the steps for using COM interoperability. We will also explore how to debug our Visual Basic .NET RTD server from the Visual Studio .NET Integrated Development Environment (IDE).

Developing the Real-Time Data Server

To begin our Visual Basic .NET RTD server, we will first need to start Visual Studio .NET Beta 2 and create a new Visual Basic .NET Class Library project, naming it VBNetRTD. Once that is done, we will definitely want to rename the default class that Visual Basic .NET creates for us from **Class1** to something more descriptive like StockQuote. We will also need to add a reference to the Microsoft Excel 10.0 Object Library; to do this, click **Add Reference** on the **Project** menu, click **Add Reference**, and click the **COM** tab. In the list, double-click **Microsoft Excel 10.0 Object Library**, and click **OK**.



Component Name	TypeLib Ver...	Path
Microsoft Excel 10.0 Object Li...	1.4	C:\Program Files\Microsoft O...
Microsoft Excel 5.0 Object Lib...	1.0	C:\Program Files\Microsoft O...
Microsoft Exchange Event Ser...	1.0	C:\Program Files\Common Fil...
Microsoft Exchange Instant M...	1.0	C:\Program Files\Messenger\...
Microsoft FlexGrid Control 6.0...	1.0	C:\WINDOWS\System32\Msfl...
Microsoft Forms 2.0 Object Li...	2.0	C:\WINDOWS\System32\FM...
Microsoft Forms 2.0 Object Li...	2.0	C:\WINDOWS\System32\FM...
Microsoft FrontPage 5.0 Page...	5.0	C:\PROGRA~1\MICROS~3\O...
Microsoft FrontPage 5.0 Web ...	5.0	C:\PROGRA~1\MICROS~3\O...
Microsoft Graph 10.0 Object L...	1.4	C:\Program Files\Microsoft O...
Microsoft H323 Service Provid...	1.0	C:\WINDOWS\System32\h32...
Microsoft Help Data Services ...	1.0	C:\Program Files\Common Fil...

Component Name	Type	Source
Microsoft Excel 10.0 Object Library	COM	C:\Program Files\Microsoft Offi...

Figure 1. The Add Reference dialog box (click image to see larger picture)

After clicking **OK**, you will get the following message: "Could not find a primary interop assembly for the COM component 'Microsoft Excel 10.0 Object Library.' A primary interop assembly is not registered for this type library. Would you like to have a wrapper generated for you?" You will need to click **Yes** to this dialog and allow Visual Basic .NET to create the interoperability assembly; otherwise, you won't be able to implement the **IRTDServer** interface. Once this is done, a new assembly is created for you with the runtime callable wrappers that will be used to call the COM objects from the managed .NET environment. For more information and details on the import process and alternatives, see Importing a Type Library as an Assembly in the .NET Framework Developer's Guide.

Now that we have a reference to the Excel type library, we will build our RTD server component. In order to allow Excel types to be referenced directly without qualification—as well as some other namespaces that we'll need for our project—we'll add the following imports to the header of the class:

```
Imports Excel                              '<-- Excel object library
Imports System.Timers                      '<-- For our timers.
Imports System.Runtime.InteropServices     '<-- For our interop attributes.
Imports System.Collections                 '<-- For our collections of quotes.
```

While it isn't required to use the **Imports** statements in our class, it greatly simplifies variable declaration because we don't have to fully qualify our object by namespace (for example, we can use **Timer** instead of **System.Timers.Timer**). Note that, unlike Visual Basic 6.0, simply adding a reference to the library will not allow us to do this. Instead, the reference includes the assembly in the compilation; in order to allow unqualified reference to types, the namespace (the closest equivalent in .NET to a type library) must be specifically imported in the very beginning of the class file. The **Imports** statement must precede any other declarations except **Option** statements. The **Imports** statement is not a replacement for adding references to required components.

Now that we have set up the imports, we will need to start implementing the **IRTDServer** interface on our StockQuote class. Like Visual Basic 6.0, this is accomplished using the **Implements** keyword and specifying the interface that we want to implement (**IRTDServer**).

We'll also need to add some attributes to the class for use by COM Interop services: **ProgID** and **ComVisible**.

The **ProgID** attribute allows you to specify the ProgID that will be used when the class is registered with COM. This is not required and, if not specified on the COM-registered class, the .NET Framework COM registration utility (regasm.exe) will create the ProgID for you. However, we want to control the ProgID and have it appear as Stock.Quote since it will be easier for us to train our end users to use it that way.

The **ComVisible** attribute can also be applied to the assembly in the AssemblyInfo.vb file (to make all public classes visible or invisible to COM) or to individual classes (to make specific classes visible or invisible to COM). In our project, we will apply this only to the classes we want registered with COM (our RTD server class).

The attributes that are associated with a class must also be on the same line of code as the class declaration (this holds true for subroutines and functions as well), so we will use the line continuation characters to make them a little more readable. Also, since our class will be used by COM, we'll need to make sure that we have a default constructor that takes no arguments. By the time we're done, our class looks like this:

```
Imports Excel                            '<-- Excel's object library.
Imports System.Timers                    '<-- For our timers.
Imports System.Runtime.InteropServices   '<-- For our interop attributes.
Imports System.Collections               '<-- For our collections of quotes.

<ComVisible(True), ProgId("Stock.Quote")> _
Public Class StockQuote
    Implements IRtdServer

    'Default constructor with no arguments.
    'Required for COM interop.
    Public Sub New()

    End Sub

End Class
```

Now that we have the class set up, we need to add the interface members for the **IRTDServer** interface. While we could type them in manually, we will allow the Visual Basic .NET IDE to create the procedure definitions for us. The process for this is exactly the same as in Visual Basic 6.0; select the interface from the **Class Name** list on the left side of the code editor, and select each procedure from the **Method Name** list on the right side of the code editor. This allows us to quickly and easily define all of our procedures with the proper syntax and types.

One significant difference between Visual Basic 6.0 and Visual Basic .NET that is readily apparent is the declaration of implemented methods. In Visual Basic 6.0, this would be a private method and named `InterfaceName_MethodName`; for example, the RTD server's **Heartbeat** method would be declared `Private Function IRTDServer_Heartbeat() as Long`. In Visual Basic .NET, however, the implemented method is specified using the **Implements** keyword with the method and specifying the interface and member that the function is implementing: `Public Function Heartbeat() As Integer Implements Excel.IRtdServer.Heartbeat`. By specifying the implemented function as **Public**, it will appear on both our RTD server's default interface as well as on the **IRTDServer** interface. If we don't want the function to be a member of the class's interface, we would declare it as **Private** instead.

Keeping Track of RTD Data

Before we move too fast, though, we need to give some thought to a couple of things that will support our RTD server. First, we need to determine how we will keep track of the state for our stock quotes—the topic IDs, the ticker symbol, and the data—as well as how the RTD server keeps track of the topics. To do this, we'll take advantage of a new feature of Visual Basic .NET—inner classes. An inner class is one that is declared within another class, and by declaring it as **Private**, it is accessible only from that class. Our class will have properties for the topic ID, ticker symbol, and last trade, as well as a method for updating the value. Since we will need to have a ticker symbol for this class to do anything, we will make this a required argument in the class's constructor and also use this constructor to set an initial opening price for the stock. When this is done, our new class should have the following code in our inner class:

```
Private Class RTDData
    'Private variables for the properties.
    Private m_strTicker As String
    Private m_sngLast, m_sngOpen As Single
    Private m_intTopicID As Integer = -1

    'Constructor.
    Public Sub New(ByVal NewTicker As String)
        Dim rdm As Random = New Random()
        'Set ticker and topic ID.
        m_strTicker = NewTicker
        'Use a random number for the opening price.
        m_sngOpen = rdm.NextDouble() * 100
        'Set the last price to the opening price.
        m_sngLast = m_sngOpen
    End Sub

    Public ReadOnly Property Ticker() As String
        Get
            Return m_strTicker
        End Get
    End Property

    Public ReadOnly Property Last() As Single
        Get
            Return m_sngLast
        End Get
    End Property

    Public ReadOnly Property Open() As Single
        Get
            Return m_sngOpen
        End Get
    End Property

    Public Property TopicID() As Integer
        Get
            Return m_intTopicID
        End Get
        Set(ByVal Value As Integer)
            m_intTopicID = Value
        End Set
    End Property

    Public Sub Update(ByVal rdm As Random)
        'Get the price change.
        Dim sngPriceChange = rdm.NextDouble() * 5
        If rdm.Next(1, 10) < 5 Then
            'Drop in price.
            m_sngLast -= sngPriceChange
        Else
            'Increase in price.
            m_sngLast += sngPriceChange
        End If
    End Sub
End Class
```

There are a couple of things that are new to Visual Basic .NET worth pointing out here. First, when we declared m_sngLast and m_sngOpen, we only specified the data type (**Single**) once. In Visual Basic 6.0, m_sngLast would be a variable of type **Variant**; only m_sngOpen would be of type **Single**. Visual Basic .NET changes this behavior—both are **Single** values. Also note the structure of the constructor (`Public Sub New`). A constructor is called when the class is created—like `Class_Initialize`—but can take arguments that are passed in from the creator when the class is instantiated and can be overloaded, allowing for multiple constructors. If a constructor is not specified, Visual Basic .NET will create a default constructor with no arguments for you. However, if you define any constructor at all, Visual Basic .NET will not create this default constructor. Therefore, by defining only a constructor that takes arguments, these arguments (in our case, there's only one, but there can be several) must be supplied to instantiate the class. Last, but not least, we also have the ability to do arithmetic operator shortcuts (**+=** and **-=**), as we do in the Update subroutine with m_sngLast. These arithmetic operator shortcuts allow for easier expression of addition and subtractions; for example the statement `x += 1` is equivalent to `x = x + 1`. While this doesn't add any functionality, it is more expressive, easier to write, and more consistent with other modern programming languages.

Once we have our data storage class in place, we'll also need to have a way to keep track of all of the RTDData classes that are created, we'll use an old-fashioned Visual Basic **Collection**, which is still supported in Visual Basic .NET. This will be declared as a private member variable in our StockTicker class and instantiated in the declaration. We'll also need to declare a **Timer** to simulate our stock "updates" as well as our reference to the Excel-supplied **IRTDUpdateEvent** class. Like our collection, these will be private member variables in our StockTicker class and we will call them m_tmrUpdate and m_xlRTDUpdate, respectively. Since we're interested in the timer's **Elapsed** event, we'll also declare the timer using the **WithEvents** keyword. By the time we have finished this, the Declarations section of the StockQuote class should be similar to the following:

```
Private m_xlRTDUpdate As Excel.IRTDUpdateEvent
Private WithEvents m_tmrTimer As Timer
Private m_colRTDData As Collection = New Collection()
```

Implementing the RTD Server

Now we are finally ready to begin implementing the RTD server. There are several things that we need to do, but we will try to take them on one bit at a time. First, we will do the easiest method on the interface: **Heartbeat**. From this function, we simply need to return a 1, which lets Excel know that our RTD server is still "alive". Note that to do this, we will use the new **Return** keyword. The **Return** keyword will return the value specified from the current function just as setting the value of the function does (for example, Heartbeat = 1). However, the **Return** keyword will then exit the function immediately, returning control to the calling procedure.

```
Public Function Heartbeat() As Integer _
        Implements Excel.IRtdServer.Heartbeat
    Return 1
End Function
```

Next, we will implement the **ServerStart** method. First, we need to set m_xlRTDUpdate to have a reference to the **CallbackObject** object. We'll also instantiate and prepare the timer that we will use to simulate updates in our sample component and get it set up, though we won't start it until later. When we instantiate it, we will use its constructor to set the default interval of 2000 milliseconds (2 seconds) and then set the **AutoReset** property to **True**, which causes the timer to continually fire. Finally, we also need to return 1 from the method to let Excel know that all is well. While we're at it, we'll also implement ServerTerminate, which will clean up all of our references. Notice that we no longer need to use the **Set** keyword when working with object references. If, out of habit, you type **Set** anyway, don't worry—Visual Basic .NET will remove it for you.

```
Public Function ServerStart(ByVal CallbackObject As Excel.IRTDUpdateEvent) _
        As Integer Implements Excel.IRtdServer.ServerStart
    'Hold a reference to the callback object.
    m_xlRTDUpdate = CallbackObject
    'Create the time with a 2000 millisecond interval.
    m_tmrTimer = New Timer(2000)
    m_tmrTimer.AutoReset = True
    'All is well, return 1.
    Return 1
End Function

Public Sub ServerTerminate() Implements Excel.IRtdServer.ServerTerminate
    'Clear the RTDUpdateEvent reference.
    m_xlRTDUpdate = Nothing
    'Make sure the timer is stopped.
    If m_tmrTimer.Enabled Then
        m_tmrTimer.Stop()
    End If
    m_tmrTimer = Nothing
End Sub
```

Now we need to work with the RTD server's **ConnectData** and **DisconnectData** methods. These methods will be the Excel way to let the RTD server know when a user has requested a new quote, as well as when they are done. In **ConnectData**, we'll check to see if we already have the related **RTDData** class for the requested ticker in our collection and create it if we do not. The new Visual Basic .NET structured exception handling now gives us the ability to implement this using complex error handling schemes that were difficult, if not impossible, with Visual Basic 6.0. Structured exception handling uses TryCatch code blocks and allows error handling schemes to be nested without difficulty—a significant improvement over Visual Basic 6.0 On Error GoTo error handling. We'll also need to check the input to see if the user requested the stock's opening price or the price at the last trade through arguments in the Excel RTD function. If there is an unexpected error in the routine, we will want to display some text indicating that there was an error to the user. In **DisconnectData**, we will simply remove the **RTDData** class from our collection if we do not need it anymore.

```
Public Function ConnectData(ByVal TopicID As Integer, _
        ByRef Strings As System.Array, _
        ByRef GetNewValues As Boolean) As Object _
        Implements Excel.IRtdServer.ConnectData
    Dim strValue, strTicker As String
    Dim objRTDData As RTDData

    'Make sure that the timer is started.
    If Not m_tmrTimer.Enabled Then
        m_tmrTimer.Start()
    End If
    GetNewValues = True
    Try
        strTicker = UCase(Strings(0))       'First argument is the ticker.
        strValue = LCase(Strings(1))        'Second argument is the type.

        'Check the value and act appropriately.
        If strValue = "last" Then
            'They want the last value.
            'Check if the data object was created.
            Try
                objRTDData = m_colRTDData.Item(strTicker)
            Catch
                'Item wasn't found - create.
                objRTDData = New RTDData(strTicker)
                'Add to collection.
                m_colRTDData.Add(objRTDData, strTicker)
            End Try
            If objRTDData.TopicID = -1 Then
                'We want this one's topic ID for later updates.
                objRTDData.TopicID = TopicID
            End If
            Return CObj(objRTDData.Last)
        ElseIf strValue = "open" Then
            'They want the opening price.
            'Check if the data object was created.
            Try
                objRTDData = m_colRTDData.Item(strTicker)
            Catch
                'Item wasn't found - create.
                objRTDData = New RTDData(strTicker)
                'Add to collection.
                m_colRTDData.Add(objRTDData, strTicker)
            End Try
            Return CObj(objRTDData.Open)
        Else
            'Unrecognized value requested.
            'Tell the user.
            Return "Unrecognized value requested"
        End If

    Catch
        'Any unexpected error.
        Return "ERROR IN QUOTE"
    End Try
End Function
```

```
Public Sub DisconnectData(ByVal TopicID As Integer) _
        Implements Excel.IRtdServer.DisconnectData
    'User no longer wants the quote.
    'Loop over the quotes and try to find it.
    Dim objRTDData As RTDData
    For Each objRTDData In m_colRTDData
        If objRTDData.TopicID = TopicID Then
            'Found it ... remove it.
            m_colRTDData.Remove(objRTDData.Ticker)
        End If
    Next
    'Stop the timer if we are done.
    If m_colRTDData.Count = 0 And m_tmrTimer.Enabled Then
        m_tmrTimer.Stop()
    End If
End Sub
```

Before we get too far ahead of ourselves and implement the **RefeshData** method, we need to write the code that will do our random updates. As in Visual Basic 6.0, we will select the member variable from the **ClassName** list on the left side of the code editor and the **Elapsed** event from **Method Name** list on the right side of the code editor. This will automatically create the event definition for us to code. This method will simply loop over the items that are currently in the collection of **RTDData** items and call the **Update** method on each, passing a shared **Random** class to the method. We will also call UpdateNotify to let Excel know that we have new data ready for display and processing. Notice that the syntax for the event is slightly changed in Visual Basic .NET; now we must specifically declare with the event this method handles, rather than it being implied in the declaration. While this may seem like additional typing that isn't necessary, it adds a tremendous amount of flexibility. As long as the events have the same signature, Visual Basic .NET can share event handlers between different events and objects by adding the additional events to the Handles declaration, separating each by a comma.

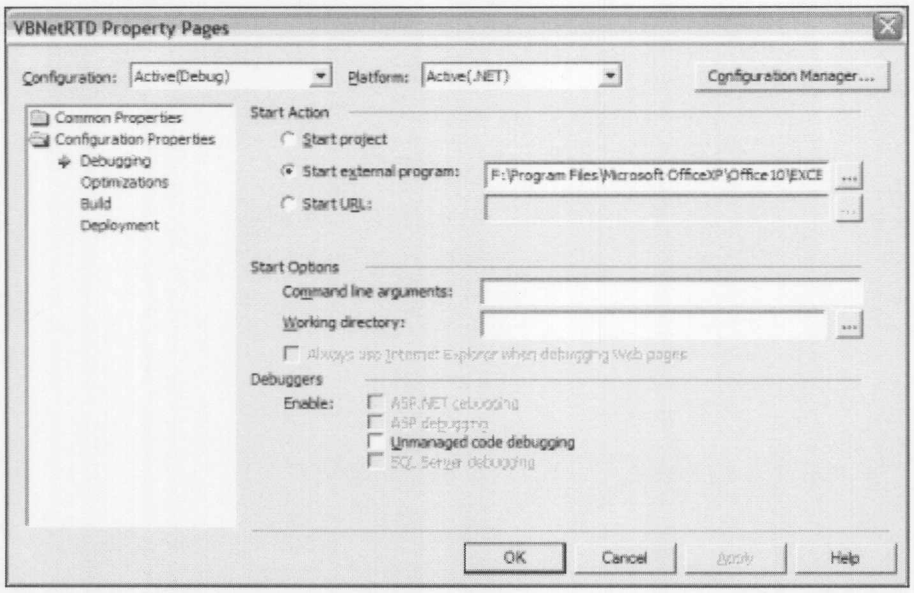
```
Private Sub m_tmrTimer_Elapsed(ByVal sender As Object, _
        ByVal e As ElapsedEventArgs) Handles m_tmrTimer.Elapsed
    Dim objData As RTDData
    'Create a shared randomizer.
    Dim rdmRandomizer As Random = New Random()
    'Call update for each stock quote.
    For Each objData In m_colRTDData
        objData.Update(rdmRandomizer)
    Next
    'Tell Excel that we have updates.
    m_xlRTDUpdate.UpdateNotify()
End Sub
```

Now, finally and at long last, we are ready to implement the **RefreshData** method on the **IRTDServer** interface. This is the key method on the interface; this is the method that Excel calls to get the latest values when it is ready for them (after, of course, our RTD server has called UpdateNotify). This method returns a two-dimensional variant array structure. The first dimension of this structure will contain the topic ID to update and the second dimension contains the new value for the topic. Unlike Visual Basic 6.0, this is not declared as Variant(); instead the return is declared as System.Array, the base array class of the .NET Framework. Also, when we declare the variable that we will be using to build the return value in the body of the method, we cannot use the **Variant** data type—Visual Basic .NET does not have this data type anymore as it is not a part of the .NET Common Type System. Instead, we will use the **Object** data type when declaring the array. If you use any other data type, you will get an error in Excel. Since the only data that changes is the last price value, these are the only topics that we will be updating—the open price never updates.

```
Public Function RefreshData(ByRef TopicCount As Integer) _
        As System.Array Implements Excel.IRtdServer.RefreshData
    Dim intItemCount As Integer
    Dim aRetVal(1, m_colRTDData.Count - 1) As Object
    Dim i As Integer
    For i = 1 To m_colRTDData.Count
        Dim curItem As RTDData = m_colRTDData.Item(i)
        If curItem.TopicID <> -1 Then
            'Update the topic with the latest value.
            aRetVal(0, i - 1) = curItem.TopicID
            aRetVal(1, i - 1) = curItem.Last
            intItemCount += 1
        End If
    Next
    'Tell Excel how many topics we updated.
    TopicCount = intItemCount
    'Return the updates.
    Return aRetVal
End Function
```

Testing and Debugging the RTD Server

Now that we have finished the code for our Excel real-time data server, it is time to test it in Excel. Instead, we need to tell Visual Studio .NET which application to start for the debug session (Excel) and then start the debug session. To do this, we will right-click on the project in the Solution Explorer window and click **Properties**. To set the debugging settings, expand the **Configuration Properties** folder and click **Debugging**. In the **Start Action** area, click **Start External Program** and enter the path to Microsoft Excel (for example, C:\Program Files\Microsoft Office\Office10\EXCEL.EXE).


VBNetRTD Property Pages

Configuration: Active(Debug)
Platform: Active(.NET)

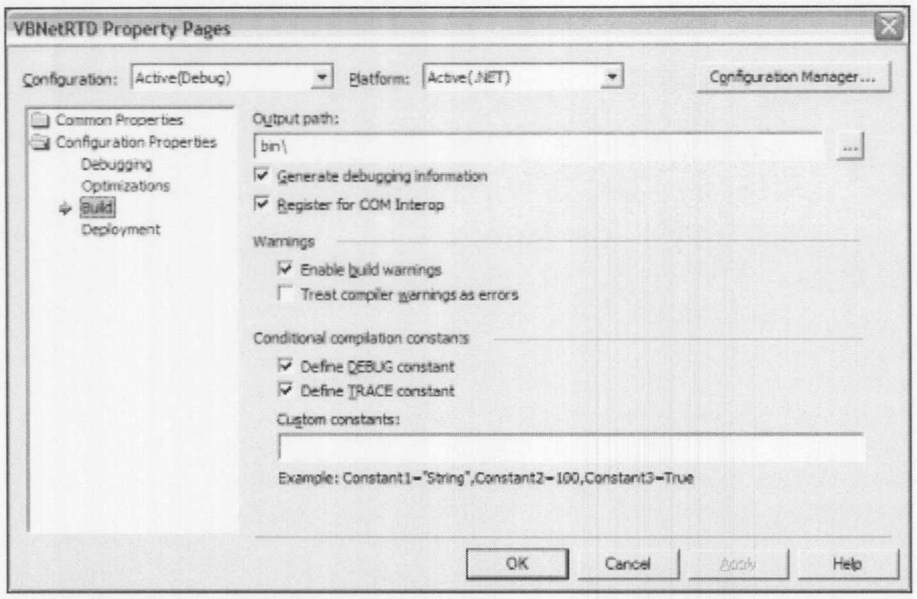
Start Action
Start external program: F:\Program Files\Microsoft OfficeXP\Office 10\EXCE

Start Options
Command line arguments:
Working directory:

Debuggers
Enable: Unmanaged code debugging

Figure 2. The Solution Property Pages dialog box: debugging options (click image to see larger picture)

We will also need to change the build settings so that the assembly is registered for COM interop by Visual Studio .NET when it gets compiled. To do this, click **Build** under the **Configuration Properties** folder and check **Register for COM Interop**. When you click **Apply**, Visual Studio .NET will display the following message: "The assembly must be strongly named if it is to be registered for COM interop. Do you want to give the assembly a strong name now?" Clicking **Yes** will generate a key for the assembly and give it a strong name; this is required for COM-visible assemblies. For more information on strong names and key files, see Creating and Using Strong-Named Assemblies in the MSDN Library.


VBNetRTD Property Pages

Configuration: Active(Debug) Platform: Active(.NET) Configuration Manager...

Common Properties
Configuration Properties
- Debugging
- Optimizations
- Build
- Deployment

Output path: bin\

[✓] Generate debugging information
[✓] Register for COM Interop

Warnings
[✓] Enable build warnings
[ ] Treat compiler warnings as errors

Conditional compilation constants
[✓] Define DEBUG constant
[✓] Define TRACE constant

Custom constants:

Example: Constant1="String",Constant2=100,Constant3=True

Figure 3. The Solution Property Pages dialog box: build options (click image to see larger picture)

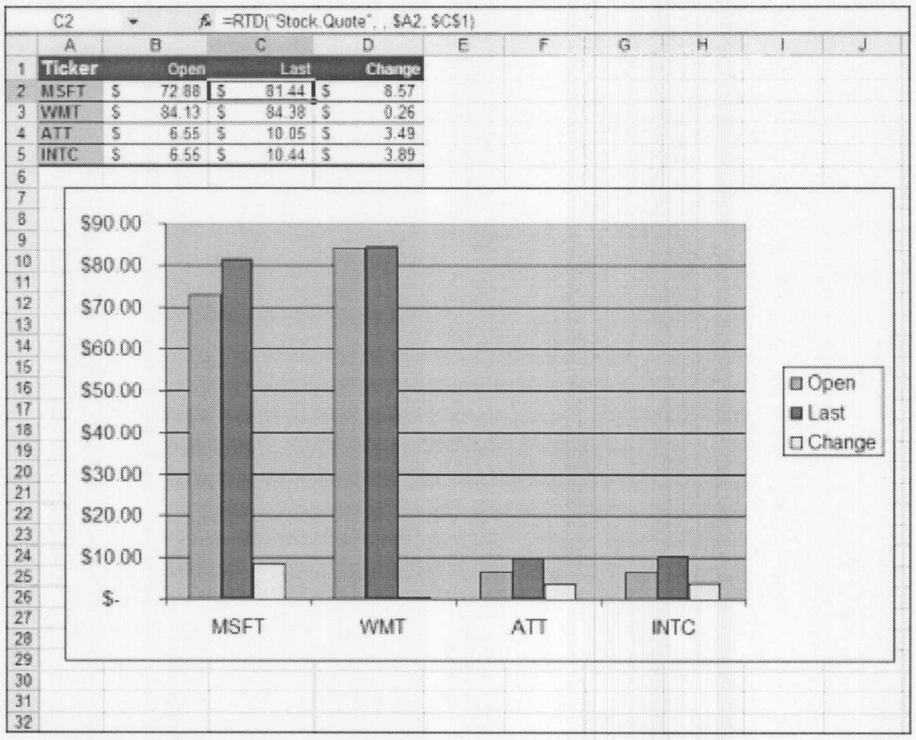
Now that we have everything set up, it is time to run the project and test the RTD server by going to the **Debug** menu and choosing **Start** (or pressing **F5**). This will compile the assembly, attach the debugger, and start Excel. To test our RTD server, we need to use the **RTD** function in Excel. Looking back at **ConnectData**, we know that we need to supply two additional arguments to the RTD server: the ticker and the type of quote (open or last). The syntax, therefore, for the **RTD** function to call our Stock Quote RTD server will be:

```
=RTD("Stock.Quote", , [Ticker], [Type (Open|Last)])
```

To test our RTD Server, we will put the following values in a blank Excel worksheet:

	A	B
1	=RTD("Stock.Quote", , "MSFT", "Open")	=RTD("Stock.Quote", , "MSFT", "Last")

This gives us a simple spreadsheet that shows the RTD components in action. However, since **RTD** is a true Excel function, we can use cell references and base calculations on the result. For example, we could integrate the results from our RTD server with charts, use AutoFill—everything that you've become used to doing in Excel. The screen shot below shows some of this.



C2 =RTD("Stock.Quote", , $A2, C1)

Ticker	Open	Last	Change
MSFT	$ 72.88	$ 81.44	$ 8.57
WMT	$ 84.13	$ 84.38	$ 0.26
ATT	$ 6.55	$ 10.05	$ 3.49
INTC	$ 6.55	$ 10.44	$ 3.89

Figure 4. Excel Workbook with graphs and calculations based on RTD (click image to see larger picture)

Conclusion

Once you understand how COM interop works in .NET, as well as some of the new features, creating a real-time data server in Visual Basic .NET is not difficult at all, although it is different from Visual Basic 6.0. This allows developers to leverage the full functionality and richness of the .NET Framework from their Office-based applications, opening the door to exciting new possibilities.